3.8 IMP++: IMP Extended with Several Features

Our overall goal in this section is to reveal and discuss the advantages and disadvantages of the various programming language semantical approaches presented in this chapter. To have a basis for the discussion, we extend the IMP language discussed previously in this chapter with several simple language features and attempt to give the extended language, called IMP++, a formal semantics following each of the approaches. In each case, we aim at reusing the existing semantics of IMP as much as possible. The additional features of IMP++ are the following:

1. A variable increment operation, `++` *Id*, whose role is to infuse side effects in expressions;
2. An output statement, `print` *AExp*, whose role is to modify the configurations (one needs to add an output "buffer", where all the output is collected);
3. Abrupt termination, both by means of an explicit `halt` statement and by means of division-by-zero, whose role is to enforce a sudden change of the evaluation context;
4. Spawning a new "thread", `spawn` (*Stmt*), which executes the given statement concurrently with the rest of the program, sharing all the variables. The role of the `spawn` statement is to test the support of the various semantic approaches for concurrent language features.

The criterion used for selecting these new features of IMP++ was twofold: on the one hand, these are quite ordinary features encountered in many languages; on the other hand, each of them exposes limitations of one or more of the conventional semantic approaches. Both IMP and IMP++ are admittedly toy languages; however, if a certain programming language semantical style has difficulties in supporting any of the features of IMP or any of the above IMP extensions in IMP++, or if in order to define a new feature one needs to make unrelated changes in the already existing semantics of other features, then one should most likely expect the same problems, but of course amplified, to occur in practical attempts to define real-life programming languages.

IMP++ extends IMP both syntactically and semantically. Syntactically, it adds to IMP the following four language constructs:

$$
\begin{array}{rcl}
\mathit{AExp} & ::= & \texttt{++}\,\mathit{Id} \\
\mathit{Stmt} & ::= & \texttt{print}\,(\mathit{AExp}) \\
& | & \texttt{halt} \\
& | & \texttt{spawn}\,(\mathit{Stmt})
\end{array}
$$

Semantically, in addition to defining the new language constructs above, we prefer that division-by-zero implicitly halts the program in IMP++, same like the explicit use of `halt`, but in the middle of an expression evaluation. When such an error takes place, one could also generate an error message; however, for simplicity, we do not consider error messages here, only silent termination.

Before we continue with the details of defining each of the new language features in each of the semantics, we would like to mention that there could be various ways to do these. Our goal in this section is to illustrate the lack of modularity of the various semantic styles in different language extension scenarios, and not necessarily to output the user good error messages. For example, a program that performs a division by zero simply halts its execution when the division by zero takes place and all is reported is the program state and output at that moment. One could certainly envision better ways to terminate the program; we encourage the reader to experiment with that in particular and, in general, to try to improve upon our definitions below wherever possible.

We first take the various IMP language extensions one by one, discussing what it takes to add each of them to IMP making abstraction of the other features. Moreover, to make our language design experiment more realistic, when defining each feature we pretend that we do not know what other features will be added. That is, we attempt to achieve local optima for each feature independently. Then, in Section 3.8.5, we put all the features together in the IMP++ language.

3.8.1 Adding Variable Increment

Like in several main-stream programming languages, `++`x increments the value of x in the state and evaluates to the incremented value. This way, the increment operation makes the evaluation of expressions to now have side effects.

Big-Step SOS

The big-step operational semantics is the most affected by the inclusion of side effects in expressions, because the previous triples $\langle a, \sigma\rangle \Downarrow \langle i\rangle$ and $\langle b, \sigma\rangle \Downarrow \langle t\rangle$ need to change to four-tuples of the form $\langle a, \sigma\rangle \Downarrow \langle i, \sigma'\rangle$ and $\langle b, \sigma\rangle \Downarrow \langle t, \sigma'\rangle$, to account for collecting the possible side effects generated by the evaluation of expressions (note that the evaluation of Boolean expressions, because of `<=`, can also have side effects). The big-step semantics of all the language constructs needs to change as well. For example, the semantics of `/` changes as follows:

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1, \sigma_1\rangle \qquad \langle a_2, \sigma_1\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1 \texttt{ / } a_2, \sigma\rangle \Downarrow \langle i_1 /_{Int} i_2, \sigma_2\rangle} \quad , \quad \text{where } i_2 \neq 0 \text{ (BigStep-Div-Left-to-Right)}$$

$$\frac{\langle a_1, \sigma_2\rangle \Downarrow \langle i_1, \sigma_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle i_2, \sigma_2\rangle}{\langle a_1 \texttt{ / } a_2, \sigma\rangle \Downarrow \langle i_1 /_{Int} i_2, \sigma_1\rangle} \quad , \quad \text{where } i_2 \neq 0 \text{ (BigStep-Div-Right-to-Left)}$$

The rules above make an attempt to capture the intended nondeterministic evaluation strategy of the division operator. We will shortly explain why they fail to fully capture the desired behaviors.

Let us next include the big-step semantics of the increment operation; once all the changes to the existing semantics of IMP are applied, the big-step semantics of increment is straightforward:

$$\langle \texttt{++}\, x, \sigma\rangle \Downarrow \langle \sigma(x) +_{Int} 1, \sigma[(\sigma(x) +_{Int} 1)/x]\rangle \qquad \text{(BigStep-Inc)}$$

Indeed, the problem with big-step is not to define the semantics of variable increment, but what it takes to be able to do it. One needs to redefine configurations as explained above and, consequently, to change the semantics of almost all the already existing features of IMP to use the new configurations. This, and other features defined later on, show how non-modular big-step semantics is.

Moreover, the addition of side-effects makes the originally intended evaluation strategies of the various expression constructs important. Indeed, for demonstration purposes we originally wanted `+` and `/` to be non-deterministic (i.e., to evaluate their arguments stepwise non-deterministically, possibly interleaving their evaluations), while `<=` to be left-right sequential. These different evaluation strategies can now lead to different behaviors, which, unfortunately, cannot be naturally captured by the big-step, or "natural" semantics. While we can still capture some limited degree of non-determinism as we showed above with the definition of `/`, namely we can choose which of the subexpressions to evaluate first, we cannot define the full non-deterministic strategy (unless we make radical changes to the definition, such as working with sets of values instead of values, which significantly complicates the big-step definition and still fails to capture the non-deterministic

behaviors — it would only capture the non-deterministic evaluation results). To see how the non-deterministic "choice" evaluation strategy in big-step semantics fails to capture all the desired behaviors, consider the the expression "$\texttt{++}x \texttt{ / (} \texttt{++}x \texttt{ / } x\texttt{)}$" with x initially 1. This expression can only evaluate to 1, 2 or 3 under non-deterministic choice strategy like we get in big-step, but it can also evaluate to 0 and even perform a division-by-zero under a fully non-deterministic evaluation strategy, like we will correctly get using the small-step operational semantic approaches.

Big-step semantics not only misses behaviors due to its lack of support for non-deterministic evaluation strategies, like shown above, but also hides misbehaviors that it, in principle, detects. For example, the expression "$1 \texttt{ / (} x \texttt{ / } \texttt{++}x\texttt{)}$" (assume $x > 0$) can either evaluate to 1 or perform an erroneous division by zero. If one searches for all the possible evaluations of a program containing such an expression using the big-step semantics in this section, one will only see the behavior where this expression evaluates to 1; one will never see the erroneous behavior where the division by zero takes place. This will be fixed shortly when we modify the big-step semantics to support abrupt termination. However, without modifying the semantics, the language designer using big-step semantics may wrongly think that the program is correct. Contrast that with the small-step semantics, which, even when one does not add support for abrupt termination, still detects the wrong behavior by getting stuck on the configuration obtained right before the division by zero.

Additionally, as explained in Section 3.2.3, the new configurations may also interfere with the rewriting infrastructure when one wants to execute big-step definitions using rewriting. Indeed, one needs to remove resulting rules that lead to non-termination, such as the rewriting rules of the form $R \to R$ corresponding to big-step sequents $R \Downarrow R$ where R are result configurations (e.g., $\langle i, \sigma \rangle$ with $i \in \mathit{Int}$ or $\langle t, \sigma \rangle$ with $t \in \{\texttt{true}, \texttt{false}\}$). Of course, we do not use this argument against big-step operational semantics (its serious break of modularity is already sufficient to disqualify big-step in the competition for an ideal language definitional framework), but rather as a warning message to the interested reader who wants to use rewriting logic to define and possibly to use rewriting engines (like Maude) to execute big-step operational semantics.

Exercise 89. *Add variable increment to* IMP*, using big-step SOS:*

1. *Write the complete big-step SOS as a proof system;*
2. *Translate the proof system at 1. into a rewriting logic theory, like in Figure 3.8;*
3. ☆ *Implement in Maude the rewriting logic theory at 2. above, like in Figure 3.9. To test it, add to the* IMP *programs in Figure 3.4 the following two:*

```
op sum++Pgm : -> Pgm .
eq sum++Pgm = (
     vars m, n, s ;
     n := 100 ;
     while (++ m <= n) do (s := s + m)
) .

op nondetPgm : -> Pgm .
eq nondetPgm = (
     vars x ;
     x := 1 ;
     x := ++ x / (++ x / x)
) .
```

The first program should have only one behavior, so, for example, both Maude commands below

```
rewrite < sum++Pgm > .
search  < sum++Pgm > =>! Cfg:Configuration .
```

should show the same result configuration, `< m |-> 101 & n |-> 100 & s |-> 5050 >`*. The second program should have (only) three different behaviors under big-step semantics; the first command below will show one of the three behaviors, but the second will show all three of them:*

```
rewrite < nondetPgm > .
search  < nondetPgm > =>! Cfg:Configuration .
```

The three behaviors captured by the big-step semantics result in configurations `< x |-> 1 >`*,* `< x |-> 2 >`*, and* `< x |-> 3 >`*. As explained above, the big-step semantics so far should not be able to expose the behaviors in which* `x` *is* `0` *and in which a division-by-zero takes place.*

Type System using Big-Step SOS

The typing policy of variable increment is the same as that of variable lookup, that is, the incremented variable types to an integer provided it has been declared. All we need is to add the following typing rule for increment to the already existing typing rules in Figure 3.10:

$$(xl, x, xl') \vdash x : int \qquad \text{(BigStepTypeSystem-Inc)}$$

Exercise 90. *Type* IMP *extended with variable increment:*

1. *Translate the big-step rule above into a corresponding rewriting logic rule that can be added to the already existing rewrite logic theory in Figure 3.11 corresponding the type system of* IMP*;*
2. ☆ *Implement the above in Maude, extending the implementation in Figure 3.12. Test it on the two additional programs in Example 89.*

Small-Step SOS

Including side effects in expressions is not as bad in small-step semantics as in big-step semantics, because, as discussed in Section 3.3, in small-step SOS one typically uses sequents whose left and right configurations have the same structure even in cases where only some of the configuration components change (e.g., one typically uses sequents of the form $\langle a, \sigma\rangle \to \langle a', \sigma\rangle$ instead of $\langle a, \sigma\rangle \to \langle a'\rangle$); thus, expressions, like any other syntactic categories including statements, can seamlessly modify the state if they need to. However, since we deliberately did not anticipate the inclusion of side effects in expression evaluation, we still have to go back through the existing definition and modify *all* the rules involving expressions to propagate the side effects. For example, the small-step rule

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma\rangle}{\langle a_1 \ / \ a_2, \sigma\rangle \to \langle a_1' \ / \ a_2, \sigma\rangle}$$

for the first argument of `/` does not apply anymore when the next step in a_1 is an increment operation (since the state σ changes), so it needs to change to

$$\frac{\langle a_1, \sigma\rangle \to \langle a_1', \sigma_1\rangle}{\langle a_1 \ / \ a_2, \sigma\rangle \to \langle a_1' \ / \ a_2, \sigma_1\rangle}$$

Of course, all these changes due to side-effect propagation would have not been necessary if we anticipated that side effects may be added to the language, but the entire point of this exercise is to study the strengths of the various semantic approaches without knowing what comes next.

Once all the changes are applied, one can define the small-step semantics of the increment operation almost identically to its big-step semantics:

$$\langle \texttt{++}\, x, \sigma \rangle \to \langle \sigma(x) +_{Int} 1, \sigma[(\sigma(x) +_{Int} 1)/x] \rangle \qquad \text{(SmallStep-Inc)}$$

Exercise 91. *Same as Exercise 89, but for small-step SOS instead of big-step SOS.*
☆ *Make sure that the small-step definition in Maude exhibits all five behaviors of program* `nondetPgm` *defined in Exercise 89 (the three behaviors exposed by the big-step definition in Maude in Exercise 89, plus one where* `x` *is* `0` *and one where the program gets stuck right before a division by zero).*

MSOS

In MSOS, one can define the increment modularly:

$$\texttt{++}\, x \xrightarrow{\{\mathsf{state}=\sigma,\mathsf{state'}=\sigma[(\sigma(x)+_{Int}1)/x],\ldots\}} \sigma(x) +_{Int} 1 \qquad \text{(MSOS-Inc)}$$

No other rule needs to be changed, because MSOS already assumes that, unless otherwise specified, each rule propagates all the configuration changes in its condition(s).

Exercise 92. *Same as Exercise 89, but for MSOS instead of big-step SOS.*
☆ *Make sure that the MSOS definition in Maude, like the small-step definition in Maude in Exercise 91, exhibits all five behaviors of program* `nondetPgm` *defined in Exercise 89.*

Evaluation Contexts

Evaluation contexts can also be elegantly used to define increment modularly:

$$\langle c, \sigma \rangle [\texttt{++}\, x] \to \langle c, \sigma[(\sigma(x) +_{Int} 1)/x] \rangle [\sigma(x) +_{Int} 1] \qquad \text{(RSEC-Inc)}$$

No other rule needs to change, because the language-specific rules of a reduction semantics with evaluation contexts definition are unconditional, each rule matching and modifying only its related part of the configuration. In other words, each rule application propagates all the configuration changes due to the applications of other rules. Note that the only conditional rule that may be possible, the characteristic rule which is language-independent and says that a reduction can be applied in any appropriate context, does not inhibit the propagation of side-effects either.

Exercise 93. *Same as Exercise 89, but for reduction semantics with evaluation contexts instead of big-step SOS.*
☆ *Like for the Maude definitions of small-step and MSOS above, make sure that the resulting Maude definition exhibits all five behaviors of program* `nondetPgm` *defined in Exercise 89.*

CHAM

The chemical abstract machine can also define the increment modularly:

$$\{\!|\texttt{++}x \rightharpoonup c|\!\}\ \{\!|x \mapsto i \triangleright \sigma|\!\} \to \{\!|i +_{Int} 1 \rightharpoonup c|\!\}\ \{\!|x \mapsto i +_{Int} 1 \triangleright \sigma|\!\} \qquad \text{(CHAM-Inc)}$$

Exercise 94. *Same as Exercise 89, but for CHAM instead of big-step SOS.*
☆ *Like for the Maude definition of big-step SOS and unlike for the Maude definitions of small-step SOS, MSOS and reduction semantics with evaluation contexts above, the resulting Maude definition can only exhibit three behaviors (instead of five) of the program* `nondetPgm` *in Exercise 89. This limitation is due to our decision (in Section 3.6.1) to only heat on non-results and cool on results when implementing CHAMs into Maude. This way, the resulting Maude specifications are executable at the expense at losing some of the behaviors due to non-deterministic evaluation strategies.*

3.8.2 Adding Output

The semantics of the output statement `print` (a) is that a is first evaluated to some integer i, which is then collected in an "output buffer". By an output buffer we here mean some list structure to which one can add more elements; removing elements from that list is not allowed. In a formal language semantics, collecting the output in a list is acceptable; in implementations of the language, one will most likely want the output to be displayed as generated. Let us assume colon-separated integer lists with ϵ as identity, $\mathbf{List}^{\epsilon}_{:}\{Int\}$, and let ω, ω', ω_1, etc., range over such lists of integers. The same way we decided for notational convenience to let *State* be an alias for the map sort $\mathbf{Map}\{Id \mapsto Int\}$ (Section 3.1.2), from here on we also let *Output* alias the list sort $\mathbf{List}^{\epsilon}_{:}\{Int\}$.

There is some flexibility as to where the output buffer should be located in the configuration. One possibility is as a new top-level component in the configuration. Another possibility is as a special variable in the already existing state. The latter would require some non-trivial changes in the mathematical model of the state, so we prefer to follow the former approach in what follows. An additional argument for our choice is that sooner or later one needs to add new components to the configuration anyway, so we take this opportunity to discuss how robust/modular the various semantic styles are with regards to changes in the structure of the configuration.

Big-Step SOS

To accommodate the output buffer, the big-step SOS result configurations for statements need to change from state configurations $\langle\sigma\rangle$ with $\sigma \in State$, to pairs $\langle\sigma,\omega\rangle$, where $\omega \in Output$. That means, in particular, that the big-step SOS of all the statements implicitly needs to change, to accommodate the additional configuration output component. For example, the semantics of sequential composition needs to change from

$$\frac{\langle s_1,\sigma\rangle \Downarrow \langle\sigma_1\rangle \qquad \langle s_2,\sigma_1\rangle \Downarrow \langle\sigma_2\rangle}{\langle s_1\,;\,s_2,\sigma\rangle \Downarrow \langle\sigma_2\rangle}$$

to

$$\frac{\langle s_1,\sigma\rangle \Downarrow \langle\sigma_1,\omega_1\rangle \qquad \langle s_2,\sigma_1\rangle \Downarrow \langle\sigma_2,\omega_2\rangle}{\langle s_1\,;\,s_2,\sigma\rangle \Downarrow \langle\sigma_2,\omega_1 : \omega_2\rangle}$$

These necessary changes in big-step semantics show, again, the lack of modularity of big-step semantics. Once all the changes are applied to "update" the existing semantics to the new configurations, one can easily define the semantics of the print statement as follows:

$$\frac{\langle a, \sigma\rangle \Downarrow \langle i\rangle}{\langle \texttt{print}(a), \sigma\rangle \Downarrow \langle \sigma, i\rangle} \qquad \text{(BigStep-Print)}$$

Recall that we give the semantics of each extension of IMP in isolation, putting all the extension together only in Section 3.8.5. Therefore, we assumed that the arithmetic expression a had no side effects in the rule above.

Exercise 95. *Add output to* IMP*, using big-step SOS:*

1. *Write the complete big-step SOS as a proof system;*
2. *Translate the above into a rewriting logic theory, like in Figure 3.8;*
3. ☆ *Implement the resulting rewriting logic theory in Maude, like in Figure 3.9. To test it, use the program below (the output buffer should contain the list* `1 : 2 : ... : 100 : 101 : 5050`*):*

```
vars m, n, s ;
n := 100 ;
while (m <= n) do (
  s := s + m ;
  m := m + 1 ;
  print m
) ;
print s
```

Type System using Big-Step SOS

The typing policy of the `print` statement is straightforward, though recall that we decided to only allow it to print integers. To type programs using `print` statements we therefore add the following typing rule to the already existing typing rules in Figure 3.10:

$$\frac{xl \vdash a : int}{xl \vdash \texttt{print}\, a : stmt} \qquad \text{(BigStepTypeSystem-Print)}$$

Exercise 96. *Type* IMP *extended with output:*

1. *Translate the big-step rule above into a corresponding rewriting logic rule that can be added to the already existing rewrite theory in Figure 3.11 corresponding to the type system of* IMP*;*
2. ☆ *Implement the above in Maude, extending the implementation in Figure 3.12. Test it on the additional program in Example 95.*

Small-Step SOS

While only some result configurations of big-step SOS had to change to accommodate the `print` statement, in the case of small-step SOS all configurations corresponding to statements need to change. Implicitly, all small-step SOS rules corresponding to statements also need to change. The changes are straightforward, essentially having to just propagate the output through each statement

construct, but they are still changes and thus expose, again, the lack of modularity of small-step SOS. Here is, for example, how the small-step SOS rule for sequential composition needs to change:

$$\frac{\langle s_1, \sigma, \omega\rangle \to \langle s_1', \sigma', \omega'\rangle}{\langle s_1 \,;\, s_2, \sigma, \omega\rangle \to \langle s_1' \,;\, s_2, \sigma', \omega'\rangle}$$

The configurations corresponding to expressions do not need to change and neither do their small-step SOS rules. That is because, as already discussed, in this language design experiment we assume at each step just the current design of the language, without attempting to anticipate other features that will be possibly added in the future. For example, if functions will be added to the language, in which case expressions will also possibly affect the output through function calls, then all the small-step SOS rules need to change, including those corresponding to expressions.

Once all the changes are applied, one can give the small-step semantics of print as follows:

$$\frac{\langle a, \sigma\rangle \to \langle a', \sigma\rangle}{\langle\, \texttt{print}\, a, \sigma, \omega\rangle \to \langle\, \texttt{print}\, a', \sigma, \omega\rangle} \qquad \text{(SmallStep-Print-Arg)}$$

$$\langle\, \texttt{print}\, i, \sigma, \omega\rangle \to \langle\, \texttt{skip}\,, \sigma, \omega : i\rangle \qquad \text{(SmallStep-Print)}$$

Exercise 97. *Same as Exercise 95, but for small-step SOS instead of big-step SOS.*

MSOS

MSOS can modularly support the output extension of the language. All one needs to do is to add a new write-only attribute, say output, in the label for collecting the output and then to add the rules for the output statement as follows:

$$\frac{a \to a'}{\texttt{print}\,(a) \to \texttt{print}\,(a')} \qquad \text{(MSOS-Print-Arg)}$$

$$\texttt{print}\,(i) \xrightarrow{\{\mathsf{output'}=i,\ldots\}} \texttt{skip} \qquad \text{(MSOS-Print)}$$

Note that, since output is a write-only attribute, we only need to mention the new value that is added to the output in the label of the second rule above. If output was declared as a read-write attribute, then the label of the second rule above would have been $\{\mathsf{output} = \omega, \mathsf{output'} = \omega : i, \ldots\}$. A major implicit objective of MSOS is to minimize the amount of information that the user needs to write in each rule. Indeed, anything written by a user can lead to non-modularity and thus work against the user when changes are performed to the language. For example, if for some reason one declared output as a read-write attribute and then later on one decided to change the list construct for the output integer list from colon "$\_ : \_$" to something else, say "$\_ \cdot \_$", then one would need to change the label in the second rule above from $\{\mathsf{output} = \omega, \mathsf{output'} = \omega : i, \ldots\}$ to $\{\mathsf{output} = \omega, \mathsf{output'} = (\omega \cdot i), \ldots\}$. Therefore, in the spirit of enhanced modularity and clarity, the language designer using MSOS is strongly encouraged to use write-only (or read-only) attributes instead of read-write attributes whenever possible.

Exercise 98. *Same as Exercise 95, but for MSOS instead of big-step SOS.*

Evaluation Contexts

One needs to first change the configuration employed by our reduction semantics with evaluation contexts of IMP from $\langle s, \sigma \rangle$ to $\langle s, \sigma, \omega \rangle$, to also include the output. This change, unfortunately, generates several other changes in the existing semantics, some of them non-modular in nature. First, one needs to change the syntax of contexts from $\mathit{Context} ::= \ldots \mid \langle \mathit{Context}, \mathit{State} \rangle$ to $\mathit{Context} ::= \ldots \mid \langle \mathit{Context}, \mathit{State}, \mathit{Output} \rangle$; some change in the configuration is unavoidable in any approach (even in MSOS we added a new label), so this is not problematic. What is inconvenient is that the rules for variable lookup and for assignment need the complete configuration, so they have to change from

$$\begin{array}{rcl} \langle c, \sigma \rangle[x] & \to & \langle c, \sigma \rangle[\sigma(x)] \\ \langle c, \sigma \rangle[x := i] & \to & \langle c, \sigma[i/x] \rangle[\texttt{skip}] \end{array}$$

to

$$\begin{array}{rcl} \langle c, \sigma, \omega \rangle[x] & \to & \langle c, \sigma, \omega \rangle[\sigma(x)] \\ \langle c, \sigma, \omega \rangle[x := i] & \to & \langle c, \sigma[i/x], \omega \rangle[\texttt{skip}] \end{array}$$

Once the changes above are applied, one is ready for adding the evaluation context for the print statement as well as its reduction semantics rule as follows:

$$\mathit{Context} ::= \ldots \mid \texttt{print}\,(\mathit{Context})$$
$$\langle c, \sigma, \omega \rangle[\,\texttt{print}\ i\,] \to \langle c, \sigma, \omega : i \rangle[\,\texttt{skip}\,] \qquad \text{(RSEC-Print)}$$

Other possibilities to add the output to the configuration and to give the reduction semantics with evaluation contexts of the above language features in a way that appears to be more modular (but which yields other problems) are discussed in Section 3.9.

Exercise 99. *Same as Exercise 95, but for reduction semantics with evaluation contexts instead of big-step SOS.*

Exercise* 100. *Consider the hypothetical framework combining MSOS with reduction semantics with evaluation contexts proposed in Exercise 56, and in particular the* IMP *semantics in such a framework in Exercise 60, its rewriting logic embeddings in Exercises 67, 72, and 77, and their Maude implementation in Exercise 82. Define the semantics of the* `print` *statement above modularly first in the framework in discussion, then using the rewriting logic embeddings, and finally in Maude.*

CHAM

All we have to do is to add a new output molecule in the top-level solution that holds the output as a list. Then to include the print statement in the semantics we first need to give its evaluation strategy by means of a heating/cooling pair and then the reaction rule that dissolves the print statement and collects its argument into the output molecule:

$$\texttt{print}\ a \curvearrowright c \rightleftharpoons a \curvearrowright \texttt{print}\,(\square) \curvearrowright c$$
$$\{\!|\,\texttt{print}\ i \curvearrowright c\,|\!\}\ \ \{\!|w|\!\} \to \{\!|\,\texttt{skip} \curvearrowright c\,|\!\}\ \ \{\!|w, i|\!\} \qquad \text{(CHAM-Print)}$$

Exercise 101. *Same as Exercise 95, but for the CHAM instead of big-step SOS.*

3.8.3 Adding Abrupt Termination

As discussed, IMP++ adds both implicit and explicit abrupt program termination. The implicit abrupt termination is given by division by zero, while the explicit abrupt termination is given by a new statement added to the language, `halt`.

For the sake of making a choice and also for demonstration purposes, in both cases of abrupt termination we would like, admittedly subjectively, the resulting configuration to have the same structure as if the program terminated normally; for example, in the case of big-step SOS, we would like the result configuration for statements to be $\langle \sigma \rangle$, where σ is the state when the program was terminated abruptly. For example, we want the programs

```
vars m, n, s ;
n := 100 ;
while true do
  if m <= n
  then (
         s := s + m ;
         m := m + 1
       )
  else halt
```

```
vars m, n, s ;
n := 100 ;
while true do
  if m <= n
  then (
         s := s + m ;
         m := m + 1
       )
  else s := s / (n / m)
```

to yield the result configuration `< m |-> 101 & n |-> 100 & s |-> 5050 >` instead of a special configuration of the form `< halting, m |-> 101 & n |-> 100 & s |-> 5050 >` or similar. Unfortunately, that is not possible in all cases without intrusively modifying the syntax of the IMP language (to "catch" the exceptional behavior and explicitly discard the additional information), since some operational styles need to make a sharp distinction between a halting configuration and a normal configuration (for propagation reasons). Adepts of those semantic styles may argue that our semantic choice above seems inappropriate, since giving more information in the result configuration, such as "this is a halting configuration", is better for all purposes than giving less information. There are, however, also reasons to always want a normal result configuration upon termination. For example, one may want to include IMP in a larger context, such as in a distributed system, where all the context wants to know about the embedded language is that it takes a statement and produces a state; IMP's internal exceptional situations are of no concern to the outer context.

We believe that there is no absolute better or worse language design, both in what regards syntax and in what regards semantics. Our task here is to make the language designer aware of the subtleties and the limitations of the various semantic approaches.

Big-Step SOS

The lack of modularity of big-step semantics will be, again, emphasized here. Let us first add the semantic definition for the implicit abrupt termination generated by division by zero. Recall that one of the big-step semantics rules for division was the following:

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \qquad \langle a_2, \sigma \rangle \Downarrow \langle i_2 \rangle}{\langle a_1 \ / \ a_2, \sigma \rangle \Downarrow \langle i_1 \ /_{Int} \ i_2 \rangle} \text{, where } i_2 \neq 0$$

We keep that unchanged, but we also add the new rule:

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle i_1 \rangle \qquad \langle a_2, \sigma \rangle \Downarrow \langle 0 \rangle}{\langle a_1 \ / \ a_2, \sigma \rangle \Downarrow \langle \texttt{error} \rangle} \qquad \text{(BigStep-Div-By-Zero)}$$

In the above rule, `error` can be regarded as a special value; alternatively, one can regard $\langle \texttt{error} \rangle$ as a special result configuration.

But what if the evaluation of a_1 or of a_2 in the above rule generates itself an error? If that is the case, then one needs to propagate that error through the division construct:

$$\frac{\langle a_1, \sigma \rangle \Downarrow \langle \texttt{error} \rangle}{\langle a_1 \texttt{ / } a_2, \sigma \rangle \Downarrow \langle \texttt{error} \rangle} \qquad \text{(BigStep-Div-Error-Left)}$$

$$\frac{\langle a_2, \sigma \rangle \Downarrow \langle \texttt{error} \rangle}{\langle a_1 \texttt{ / } a_2, \sigma \rangle \Downarrow \langle \texttt{error} \rangle} \qquad \text{(BigStep-Div-Error-Right)}$$

Note that in case one of a_1 or a_2 generates an error, then the other one is not even evaluated anymore, to faithfully capture the intended meaning of abrupt termination.

Unfortunately, one has to do the same for all the expression language constructs. This way, for each expression construct, one has to add at least as many error-propagation big-step rules as arguments that expression construct takes. Moreover, when the evaluation error reaches a statement, one needs to transform it into a "halting signal". This can be achieved by introducing a new type of result configuration, namely $\langle \texttt{halting}, \sigma \rangle$, and then adding appropriate halting propagation rules for all the statements. For example, the assignment statement needs to be added the new rule

$$\frac{\langle a, \sigma \rangle \Downarrow \langle \texttt{error} \rangle}{\langle x \texttt{ := } a, \sigma \rangle \Downarrow \langle \texttt{halting}, \sigma \rangle} \qquad \text{(BigStep-Asgn-Halt)}$$

The halting signal needs to be propagated through statement constructs, collecting the appropriate state and output. For example, the following two rules need to be included for sequential composition, in addition to the existing rule (which stays unchanged):

$$\frac{\langle s_1, \sigma \rangle \Downarrow \langle \texttt{halting}, \sigma_1 \rangle}{\langle s_1 \texttt{;} s_2, \sigma \rangle \Downarrow \langle \texttt{halting}, \sigma_1 \rangle} \qquad \text{(BigStep-Seq-Halt-Left)}$$

$$\frac{\langle s_1, \sigma \rangle \Downarrow \langle \sigma_1 \rangle,\ \langle s_2, \sigma_1 \rangle \Downarrow \langle \texttt{halting}, \sigma_2 \rangle}{\langle s_1 \texttt{ ; } s_2, \sigma \rangle \Downarrow \langle \texttt{ halting }, \sigma_2 \rangle} \qquad \text{(BigStep-Seq-Halt-Right)}$$

In addition to all the halting propagation rules, we also have to define the semantics of the explicit halt statement:

$$\langle \texttt{ halt }, \sigma \rangle \Downarrow \langle \texttt{ halting }, \sigma \rangle \qquad \text{(BigStep-Halt)}$$

Therefore, when using big-step semantics, one has to more than *double* the number of rules in order to support abrupt termination. Indeed, any argument of any language construct can yield the termination signal, so a rule is necessary to propagate that signal through the current language construct. It is hard to imagine anything worse in a language design framework. An unfortunate language designer choosing big-step semantics as her favorite language definition framework will incrementally become very reluctant to add or experiment with any new feature in her language. For example, imagine that one wants to add exceptions and break/continue of loops to IMP++.

Finally, unless one extends the language syntax, there appears to be no way to get rid of the junk result configurations $\langle \texttt{ halting }, \sigma \rangle$ that have been artificially added in order to propagate the

error or the halting signals. For example, one cannot simply add the rule

$$\frac{\langle s, \sigma\rangle \Downarrow \langle \texttt{halting}, \sigma'\rangle}{\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle}$$

because it may interfere with other rules and thus wrongly hide the halting signal; for example, it can be applied on the second hypothesis of the rule (BigStep-Seq-Halt-Right) above hiding the halting signal and thus wrongly making the normal rule (BigStep-Seq) applicable. While having junk result configurations of the form $\langle \texttt{halting}, \sigma\rangle$ may seem acceptable in our scenario here, perhaps even desirable for debugging reasons, in general one may find it inconvenient to have many types of result configurations; indeed, one would need similar junk configurations for exceptions, for break/continue of loops, for functions return, etc.

Consequently, the halting signal needs to be "caught" at the top-level of the derivation. Fortunately, IMP provides a top-level syntactic category, *Pgm*, so we can add the following rule which dissolves the potential junk configuration at the top:

$$\frac{\langle s, xl \mapsto 0\rangle \Downarrow \langle \texttt{halting}, \sigma\rangle}{\langle \texttt{vars}\ xl\ \texttt{;}\ s\rangle \Downarrow \langle \sigma\rangle} \qquad \text{(BigStep-Halt)}$$

Now we have silent abrupt termination. Note that `vars` now acts as an exception catching and dissolving for the abrupt termination signal generated by `halt` or by division-by-zero. If one does not like to use `vars` for something which has not been originally intended, then one can add an auxiliary `top` statement or program construct, then reduce the semantics of `vars` to that of `top`, and then give `top` an exception-handling-like big-step SOS, as we will shortly do for the MSOS definition of abrupt termination. This latter solution is also more general, because it does not rely on a fortunate earlier decision to have a top-level language construct.

In addition to the lack of modularity due to having to more than double the number of rules in order to add abrupt termination, the addition of all these rules can also have a significant impact on performance when one wants to execute the big-step operational semantics. Indeed, there are now four rules for division, each having the same left-hand side, $\langle a_1 \,/\, a_2, \sigma\rangle$, and some of these rules even sharing some of the hypotheses. That means that any general-purpose proof or rewrite system attempting to execute such a definition will unavoidably face the problem of searching a large space of possibilities in order to find one or all possible reductions.

Exercise 102. *Add abrupt termination as discussed above to* IMP*, using big-step SOS:*

1. *Write the complete big-step SOS as a proof system;*
2. *Translate the above into a rewriting logic theory, like in Figure 3.8;*
3. ☆ *Implement the resulting rewriting logic theory in Maude, like in Figure 3.9. To test it, execute the two programs at the beginning of Section 3.8.3. The resulting big-step SOS definition may be very slow when executed in Maude, even for small values of* `n` *(such as 2,3,4 instead of 100), which is normal (as explained above, the search space is now much larger).*

Exercise 103. *Same as Exercise 105, but using a specific* `top` *construct as explained above to "catch" the halting signal instead of the existing* `vars` *which has a different purpose in the language.*

Type System using Big-Step SOS

The typing policy of abrupt termination is clear: `halt` types to a statement and division-by-zero is ignored. Indeed, one cannot expect that a type checker, or any technique, procedure or algorithm, can detect division by zero in general: division-by-zero, like almost any other runtime property of any Turing-complete programing language, is an undecidable problem. Consequently, it is common to limit typing division to checking that the two expressions have the expected type, integer in our case, which our existing type checker for IMP already does (see Figure 3.10). We therefore only add the following typing rule for `halt`:

$$xl \vdash \texttt{halt} : stmt \qquad \text{(BigStepTypeSystem-Halt)}$$

Exercise 104. *Type* IMP *extended with abrupt termination:*

1. *Translate the big-step rule above into a corresponding rewriting logic rule that can be added to the already existing rewrite logic theory in Figure 3.11;*
2. ☆ *Implement the above in Maude, extending the implementation in Figure 3.12. Test it on the additional programs in Example 105.*

Small-Step SOS

Small-step operational semantics turns out to be almost as non-modular as big-step when it gets to defining control-intensive statements like abrupt termination. Like for big-step SOS, we need to invent special configurations to signal steps corresponding to implicit division by zero or to explicit halt statements. Unlike in the big-step SOS extension above, a single type of such special configurations suffices for small-step SOS, namely one of the form $\langle \texttt{halting}, \sigma \rangle$, where σ is the state in which the program was abruptly terminated. One could also define two types of such special configurations, one for expressions and one for statements, like in big-step SOS, but, since we decided to carry the state in the right-hand-side configuration of all sequents in our small-step SOS definitions, the only difference between the two types of configurations would be their tag, for example $\langle \texttt{error}, \sigma \rangle$ versus $\langle \texttt{halting}, \sigma \rangle$. For simplicity, in the sequel we prefer not to distinguish the configurations corresponding to the two types of abrupt termination.

With that, we can then define the small-step SOS of division by zero as follows (recall that the original SmallStep-Div rule in Figure 3.14 is "$\langle i_1 \texttt{ / } i_2, \sigma \rangle \to \langle i_1 \texttt{ /}_{Int} i_2, \sigma \rangle$ where $i_2 \neq 0$"):

$$\langle i_1 \texttt{ / } 0, \sigma \rangle \to \langle \texttt{halting}, \sigma \rangle \qquad \text{(SmallStep-Div-By-Zero)}$$

Like for the big-step SOS extension above, we have to make sure that the halting signal is correctly propagated. Here are, for example, the propagation rules through the division construct:

$$\frac{\langle a_1, \sigma \rangle \to \langle \texttt{halting}, \sigma \rangle}{\langle a_1 \texttt{ / } a_2, \sigma \rangle \to \langle \texttt{halting}, \sigma \rangle} \qquad \text{(SmallStep-Div-Error-Left)}$$

$$\frac{\langle a_2, \sigma \rangle \to \langle \texttt{halting}, \sigma \rangle}{\langle a_1 \texttt{ / } a_2, \sigma \rangle \to \langle \texttt{halting}, \sigma \rangle} \qquad \text{(SmallStep-Div-Error-Right)}$$

The two rules above are given in such a way that the semantics is faithful to the intended *computational granularity* of the defined language feature. Indeed, we want division by zero to take one

computational step to be reported as an error, as opposed to as many steps as the depth of the context in which the error has been detected; for example, a configuration containing expression `(3 / 0) / 3` should reduce to a halting configurations in one step, not in two. If one added a special error integer value and replaced the two rules above by

$$\langle \texttt{error} \,/\, a_2, \sigma \rangle \to \langle \texttt{error}, \sigma \rangle$$

$$\langle a_1 \,/\, \texttt{error}, \sigma \rangle \to \langle \texttt{error}, \sigma \rangle$$

then errors would be propagated to the top level of the program in as many small-steps as the depth of the context in which the error was generated; we do not want that.

Like in the big-step SOS above, the implicit expression errors need to propagate through the statements and halt the program. One way to do it is to generate an explicit `halt` statement and then to propagate that `halt` statement through all the statement constructs as if it was a special statement "value", until it reaches the top. However, as discussed in the paragraph above, that would generate as many steps as the depth of the evaluation contexts in which the `halt` statement is located, instead of just one step as desired. Alternatively, we can use the same approach to propagate the halting configuration through the statement constructs as we used to propagate it through the expression constructs. More precisely, we add "transition" rules from expressions to statements, like the one below (a similar one needs to be added for the conditional statement):

$$\frac{\langle a, \sigma \rangle \to \langle \texttt{halting}, \sigma \rangle}{\langle x := a, \sigma \rangle \to \langle \texttt{halting}, \sigma \rangle} \qquad \text{(SmallStep-Asgn-Halt)}$$

Once the halting signal due to a division by zero reaches the statement level, it needs to be further propagated through the sequential composition, that is, we need to add the following rule:

$$\frac{\langle s_1, \sigma \rangle \to \langle \texttt{halting}, \sigma \rangle}{\langle s_1 ; s_2, \sigma \rangle \to \langle \texttt{halting}, \sigma \rangle} \qquad \text{(SmallStep-Seq-Halt)}$$

Note that we assumed that a halting step does not change the state (used the same σ in both the left and the right configurations). One can prove by structural induction that, in our simple language scenario, that is indeed the case; alternatively, one could have equivalently used σ' instead of σ in the right-hand configurations in the rule above.

Finally, we can also generate a special halting configuration when a `halt` statement is reached:

$$\langle \texttt{halt}, \sigma \rangle \to \langle \texttt{halting}, \sigma \rangle \qquad \text{(SmallStep-Halt)}$$

At this moment, any abruptly terminated program reduces to a special configuration of the form $\langle \texttt{halting}, \sigma \rangle$. Recall that our plan, however, was to terminate the computation with a normal configuration of the form $\langle \texttt{skip}, \sigma \rangle$, regardless of whether the program terminates normally or abruptly. Like in the big-step SOS above, the naive solution to transform a step producing a halting configuration into a normal step using the rule

$$\frac{\langle s, \sigma \rangle \to \langle \texttt{halting}, \sigma \rangle}{\langle s, \sigma \rangle \to \langle \texttt{skip}, \sigma \rangle}$$

does not work. Indeed, consider a situation where the rule (SmallStep-Seq-Halt) above could apply. There is nothing to prevent the naive rule above to interfere and transform the halting

premise of (SmallStep-Seq-Halt) into a normal step producing a `skip`, which can be further fed to the conventional rule for sequential composition, (SmallStep-Seq-Arg1) in Figure 3.15, hereby continuing the execution of the program as if no abrupt termination took place.

Supposing that one wants to waste no computational steps as an artifact of the particular small-step SOS approach chosen, there seems to be no immediate way to terminate the program with a normal result configuration of the form $\langle \texttt{skip}, \sigma \rangle$ both when the program terminates abruptly and when it terminates normally. One possibility, also suggested in the case of big-step SOS discussed above and followed in the subsequent MSOS definition for abrupt termination in the sequel, is to add an auxiliary `top` language construct. With that, we can change the small-step SOS rule for variable declarations to reduce the top-level program to its body statement wrapped under this `top` construct, and then add corresponding rules to propagate normal steps under the `top` while catching the halting steps and transforming them into normal steps at the top-level. Here are four small-step SOS rules achieving this (the first rule replaces the previous one for variable declarations):

$$\langle \texttt{vars}\ xl\ \texttt{;}\ s \rangle \to \langle \texttt{top}\ s, (xl \mapsto 0) \rangle$$

$$\frac{\langle s, \sigma \rangle \to \langle s', \sigma' \rangle}{\langle \texttt{top}\ s, \sigma \rangle \to \langle \texttt{top}\ s', \sigma' \rangle}$$

$$\langle \texttt{top skip}, \sigma \rangle \to \langle \texttt{skip}, \sigma \rangle$$

$$\frac{\langle s, \sigma \rangle \to \langle \texttt{halting}, \sigma \rangle}{\langle \texttt{top}\ s, \sigma \rangle \to \langle \texttt{skip}, \sigma \rangle}$$

The rules above are such that no computational step is wasted when a halting signal takes place. Unfortunately, we had to introduce additional syntax (the `top` construct) for the sole purpose of making the semantic definition work. If one is willing to waste a computational step in order to explicitly dissolve the halting configuration replacing it by a normal one, then one can add instead the following simple small-step SOS rule, which is also the approach we are taking in the case of small-step SOS in this section, because it gives us, in our view, the best trade-off between elegance and computational intrusiveness (after all, wasting a step in a deterministic manner may be acceptable in most situations):

$$\langle \texttt{halting}, \sigma \rangle \to \langle \texttt{skip}, \sigma \rangle \qquad \text{(SmallStep-Halting)}$$

Exercise 105. *Same as Exercise 105, but for the small-step SOS above instead of big-step SOS.*

Exercise 106. *Same as Exercise 105, but use the* `top` *construct approach discussed above instead of the rule* (SmallStep-Halting)*, to avoid wasting one computational step.*

Exercise 107. *One could argue that the introduction of the halting configurations* $\langle \texttt{halting}, \sigma \rangle$ *was unnecessary, because we could have instead used the already existing configurations of the form* $\langle \texttt{halt}, \sigma \rangle$*. Give an alternative small-step SOS definition of abrupt termination which does not add special halting configurations. Can we avoid the introduction of the* `top` *construct discussed above? Comment on the disadvantages of this approach.*

MSOS

MSOS allows for a more modular semantics of abrupt termination but, unfortunately, in order to do so it needs to extend the syntax of the language with a `top` construct like above. The idea is to

use the flexible labeling mechanism of MSOS to carry the information that a configuration is in a halting status. Let us assume an additional write-only field in the MSOS labels, called halting, which is *true* whenever the program needs to halt, otherwise it is *false*[6]. Then we can add the following two natural MSOS rules that set the halting field to *true*:

$$i_1 \texttt{ / } 0 \xrightarrow{\{\mathsf{halting}'=true,\ldots\}} i_1 \texttt{ / } 0 \qquad \text{(MSOS-Div-By-Zero)}$$

$$\texttt{halt} \xrightarrow{\{\mathsf{halting}'=true,\ldots\}} \texttt{halt} \qquad \text{(MSOS-Halt)}$$

As desired, it is indeed the case now that a fact of the form $s \xrightarrow{\{\mathsf{halting}'=true,\ldots\}} s'$ is derivable if and only if $s = s'$ and the next executable step in s is either a `halt` statement or a division-by-zero expression. If one does not like keeping the syntax unchanged when an abrupt termination takes place, then one can add a new syntactic construct, say `stuck` like in [57], and replace the right-hand-side configurations above with `stuck`; that does not conceptually change anything in what follows. The setting seems therefore perfect for adding a rule of the form

$$\frac{s \xrightarrow{\{\mathsf{halting}'=true,\ldots\}} s}{s \xrightarrow{\{\mathsf{halting}'=false,\ldots\}} \texttt{skip}}$$

and declare ourselves done, because now an abruptly terminated statement terminates just like any other statement, with a `skip` statement as result and with a label containing a non-halting status. Unfortunately, that does not work, because such a rule would interfere with other rules taking statement reductions as preconditions, for example with the first precondition of the (MSOS-Seq) rule, and thus hide the actual halting status of the precondition. To properly capture the halting status, we define a top level statement construct like we discussed in the context of big-step and small-step SOS above, say `top` *Stmt*, modify the rule (MSOS-Vars) from

$$\texttt{vars}\ xl\ \texttt{;}\ s \xrightarrow{\{\mathsf{state}'=xl\mapsto 0,\ldots\}} s$$

to

$$\texttt{vars}\ xl\ \texttt{;}\ s \xrightarrow{\{\mathsf{state}'=xl\mapsto 0,\ldots\}} \texttt{top}\ s$$

to mark the top level statement, and then finally include the following two rules:

$$\frac{s \xrightarrow{\{\mathsf{halting}'=false,\ldots\}} s'}{\texttt{top}\ s \xrightarrow{\{\mathsf{halting}'=false,\ldots\}} \texttt{top}\ s'} \qquad \text{(MSOS-Top-Normal)}$$

$$\texttt{top skip} \to \texttt{skip} \qquad \text{(MSOS-Top-Skip)}$$

$$\frac{s \xrightarrow{\{halting'=true,\ldots\}} s}{\texttt{top}\ s \xrightarrow{\{\mathsf{halting}'=false,\ldots\}} \texttt{skip}} \qquad \text{(MSOS-Top-Halt)}$$

[6]Strictly speaking, MSOS requires that the write-only atributes take values from a free monoid; if one wants to be faithful to that MSOS requirement, then one can replace *true* with some letter word and *false* with the empty word.

The use of a `top` construct like above seems unavoidable if we want to achieve modularity. Indeed, we could avoid it in the small-step SOS definition of abrupt termination above at the expense of one additional small-step (to dissolve the halting configuration) because the halting configurations were explicitly, and thus non-modularly propagated through each of the language constructs, so the entire program reduced to a halting configuration whenever a division by zero or a `halt` statement was encountered. Unfortunately, that same approach does not work with MSOS (unless we want to break the modularity, like in SOS), because the syntax is not mutilated when an abrupt termination occurs. The halting signal is captured by the label of the transition. However, the label does not tell us when we are the top level in order to dissolve the halting status. Adding a new label to hold the depth of the derivation, or at least whether we are the top or not, would require one to (non-modularly) change it in each rule. The use of an additional `top` construct like we did above appears to be the best trade-off between modularity and elegance when using MSOS.

Note that, even though MSOS can be mechanically translated into SOS by associating to each MSOS attribute an SOS configuration component, the solution above to support abrupt termination modularly in MSOS is *not* modular when applied in SOS via the translation, because adding a new attribute in the label means adding a new configuration component, which already breaks the modularity of SOS. In other words, the MSOS technique above cannot be manually used in SOS to obtain a modular definition of abrupt termination in SOS.

Exercise 108. *Same as Exercise 105, but for MSOS instead of big-step SOS.*

Evaluation Contexts

For our language, reduction semantics allows very elegant, natural and modular definitions of abrupt termination, without having to extent the syntax of the original language and without adding any new reduction steps as an artifact of the approach chosen:

$$\langle c, \sigma \rangle [i \,/\, 0] \to \langle \,\texttt{skip}\,, \sigma \rangle \qquad \text{(RSOS-Div-By-Zero)}$$

$$\langle c, \sigma \rangle [\,\texttt{halt}\,] \to \langle \,\texttt{skip}\,, \sigma \rangle \qquad \text{(RSOS-Halt)}$$

Therefore, the particular evaluation context in which the abrupt termination is being generated, c, is simply discarded. This is not possible in any of the big-step, small-step or MSOS styles above, because in those styles the evaluation context c is captured by the proof context, which, like in any logical system, cannot be simply discarded. The elegance of the two rules above suggest that having the possibility to explicitly match and change the evaluation context is a very powerful and convenient feature of a language semantic framework.

Exercise 109. *Same as Exercise 105, but for reduction semantics with evaluation contexts instead of big-step SOS.*

CHAM

The CHAM semantics of abrupt termination is even more elegant and modular than that using context reduction above, because the other components of the configuration need not be mentioned:

$$i \,/\, 0 \rightharpoonup c \;\; \to \;\; \texttt{skip} \qquad \text{(CHAM-Div-By-Zero)}$$

$$\texttt{halt} \rightharpoonup c \;\; \to \;\; \texttt{skip} \qquad \text{(CHAM-Halt)}$$

Exercise 110. *Same as Exercise 105, but for the CHAM instead of big-step SOS.*

3.8.4 Adding Dynamic Threads

IMP++ adds threads to IMP, which can be dynamically created and terminated. As in the previous IMP extensions, we keep the syntax and the semantics of threads minimal. Our only syntactic extension is a `spawn` statement construct. The semantics of `spawn` S is that the statement S executes concurrently with the rest of the program, sharing and possibly concurrently modifying the same variables, like threads do. The thread corresponding to S terminates when S terminates and, when that happens, we remove the thread. Like in the previous language extensions, we want programs to terminate normally, no matter whether they make use of threads or not. For example, the programs below create 101 and, respectively, 2 new threads during their execution:

```
vars m, n, s ;
n := 100 ;
while (m <= n) do (
  spawn s := s + m ;
  m := m + 1
)
```

```
vars x ;
(
   spawn x := x + 1 ;
   spawn x := x + 10
) ;
x := x + 100
```

Yet, we want the result configurations at the end of the execution to look like before in IMP, that is, like `< skip, m |-> 101 & n |-> 100 & s |-> 5050 >` and `< skip, x |-> 111 >`, respectively, in the case of small-step SOS. We grouped the two `spawn` statements in the second program on purpose, to highlight the need for structural equivalences (this will be discussed shortly, in the context of small-step semantics). Recall that the syntax of IMP's sequential composition in Section 3.1 (see Figure 3.1) was deliberately left ambiguous, based on the hypothesis that the semantics of IMP will be given in such a way (left-to-right statement evaluation) that the syntactic ambiguity is irrelevant. Unfortunately, the addition of threads makes the above hard or impossible to achieve modularly in some of the semantic approaches.

Concurrency often implies non-determinism, which is not always desirable. For example, the first program above can also evaluate to a result configuration in which `s` is 0. This happens when the first spawned thread calculates the sum `s + m`, which is 0, but postpones writing it to `s` until all the subsequent 100 new threads complete their execution. Similarly, after the execution of the second program above, `x` can have any of the values 1, 10, 11, 100, 101, 110, 111.

Exercise 111. *Propose hypothetical executions of the second program above corresponding to any of the seven possible values of* `x`*. What is the maximum value that* `s` *can have when the first program above, as well as all its dynamically created threads, terminate? Is there some execution of the first program corresponding to each smaller value of* `s` *(but larger than or equal to 0)?*

A language designer or semanticist may find it very useful to execute and analyze programs like above in her semantics. Indeed, the existence of certain behaviors or their lack of, may suggest changes in the syntax and the semantics of the language at an early and therefore cheap design stage. For example, one may decide that one's language must be race-free on any variable except some special semaphore variables used specifically for synchronization purposes (this particular decision may be too harsh on implementations, though, which may end up having to rely on complex static analysis front-ends or even ignoring it). We make no such difficult decisions in our simple language here, limiting our goal to the bottom of the spectrum of semantic possibilities: we only aim at giving a semantics that faithfully captures all the program behaviors due to spawning threads.

We make the simplifying assumptions that we have a sequentially consistent memory and that variable read and write operations are atomic and thus unaffected by potential races. For example, if `x` is 0 in a two-threaded program where one thread is about to write 5 to `x` and the other is

about to read `x`, then the only global next steps can be either that the first thread writes 5 to `x` or that the second thread reads 0 as the value of `x`; in other words, we assume that it is impossible for the second thread to read 5 or any other value except 0 as the next small-step step in the program.

Big-Step SOS

Big-step SOS and denotational semantics are the semantical approaches which are the most affected by concurrency extensions of the base language. That is because their holistic view of computation makes it hard or impossible to capture the fine-grained execution steps and behavior interleavings that are inherent to concurrent program executions. Consider, for example, a statement of the form $\texttt{spawn}\, S_1 \texttt{ ; } S_2$ in some program state σ. The only thing we can do in big-step SOS is to evaluate S_1 and S_2 in some appropriate states and then to combine their resulting states into a result state. We do not have much room for experimentation here: we either evaluate S_1 in state σ and then S_2 in the resulting state, or evaluate S_2 in state σ and then S_1 in the resulting state. The big-step SOS rule for sequential composition already implies the former case provided that $\texttt{spawn}\, S$ can evaluate to whatever state S evaluates to, which is true and needs to be considered anyway. Thus, we can formalize the above into the following two big-step SOS rules, which can be regarded as a rather "desperate" attempt to use big-step SOS for defining concurrency:

$$\frac{\langle s, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle\, \texttt{spawn}\, s, \sigma\rangle \Downarrow \langle \sigma'\rangle} \qquad \text{(BigStep-Spawn-Arg)}$$

$$\frac{\langle s_2, \sigma\rangle \Downarrow \langle \sigma_2\rangle \qquad \langle s_1, \sigma_2\rangle \Downarrow \langle \sigma_1\rangle}{\langle\, \texttt{spawn}\, s_1 \texttt{ ; } s_2, \sigma\rangle \Downarrow \langle \sigma_1\rangle} \qquad \text{(BigStep-Spawn-Wait)}$$

Exercise 112. *Translate the two big-step SOS rules above into rewriting logic, like in Figure 3.8.* ☆ *Implement the resulting rewriting logic theory in Maude, like in Figure 3.9. To test it, execute the two programs at the beginning of Section 3.8.4. The resulting Maude big-step SOS definition may be slow on the first program for large initial values for n because, even though it does not capture all possible behaviors, it still comprises many of them. For example, searching for all the result configurations when* $n = 10$ *gives 12 possible values for s, namely 55,56,...,66. On the other hand, the second program only shows one out of the seven behaviors, namely one where x results in 111.*

Therefore, as expected, the two big-step SOS rules above capture only a limited number of the possible concurrent behaviors even for small and simple programs like the ones discussed above. One may try to change the entire big-step SOS definition of IMP to collect in result configurations all possible ways in which the corresponding fragments of program can evaluate. However, in spite of its non-modularity, there seems to be no easy way to combine, for example, the behaviors of S_1 and of S_2 into the behaviors of $\texttt{spawn}\, S_1 \texttt{ ; } S_2$.

Type System using Big-Step SOS

From a typing perspective, `spawn` is nothing but a language construct expecting a statement as argument and producing a statement as result. To type programs using `spawn` statements we therefore add the following typing rule to the already existing typing rules in Figure 3.10:

$$\frac{xl \vdash s : stmt}{xl \vdash \texttt{spawn}\, s : stmt} \qquad \text{(BigStepTypeSystem-Spawn)}$$

Exercise 113. *Type* IMP *extended with dynamic threads:*

1. *Translate the big-step rule above into a corresponding rewriting logic rule that can be added to the already existing rewrite theory in Figure 3.11 corresponding to the type system of* IMP*;*
2. ☆ *Implement the above in Maude, extending the implementation in Figure 3.12. Test it on the additional programs in Example 112.*

Small-Step SOS

Small-step semantics are more appropriate for concurrency, because they allow a finer-grain view of computation. For example, they allow to say that the next computational step of a statement of the form $\texttt{spawn}\, S_1 \texttt{ ; } S_2$ comes either from S_1 or from S_2 (which is different from saying that either S_1 or S_2 is evaluated next all the way through, like in big-step SOS). Since $\texttt{spawn}\, S_1$ is already permitted to advance by the small-step SOS rule for sequential composition, the following three small-step SOS rules achieve the desired behavioral non-determinism caused by concurrent threads:

$$\frac{\langle s, \sigma\rangle \to \langle s', \sigma'\rangle}{\langle\, \texttt{spawn}\, s, \sigma\rangle \to \langle\, \texttt{spawn}\, s', \sigma'\rangle} \qquad \text{(SmallStep-Spawn-Arg)}$$

$$\langle\, \texttt{spawn skip}\,, \sigma\rangle \to \langle\, \texttt{skip}\,, \sigma\rangle \qquad \text{(SmallStep-Spawn-Skip)}$$

$$\frac{\langle s_2, \sigma\rangle \to \langle s_2', \sigma'\rangle}{\langle\, \texttt{spawn}\, s_1 \texttt{ ; } s_2, \sigma\rangle \to \langle\, \texttt{spawn}\, s_1 \texttt{ ; } s_2', \sigma'\rangle} \qquad \text{(SmallStep-Spawn-Wait)}$$

The rule (SmallStep-Spawn-Skip) cleans up terminated threads.

Unfortunately, the three rules above are not sufficient to capture all the intended behaviors. Consider, for example, the second program at the beginning of Section 3.8.4. That program was intended to have seven different behaviors with respect to the final value of x. Our current small-step SOS misses two of those behaviors, namely those in which x results in 1 and 100, respectively.

In order for the program to terminate with $x = 1$, it needs to start the first new thread, calculate the sum `x + 1` (which is 1), then delay writing it back to x until after the second and the main threads do their writes of x. However, in order for the main thread to be allowed to execute its assignment statement, the two grouped `spawn` statements need to either terminate and become `skip` so that the rule (SmallStep-Seq-Skip) (see Figure 3.15) applies, or to reduce to only one `spawn` statement so that the rule (SmallStep-Spawn-Wait) above applies. Indeed, these are the only two rules which allow access to the second statement in a sequential composition. The first case is not possible, because, as explained, the first newly created thread cannot be terminated. In order for the second case to happen, since the first `spawn` statement cannot terminate, the only possibility is for the second `spawn` statement to be executed all the way through (which is indeed possible, thanks to the rules (SmallStep-Seq-Arg1) in Figure3.15 and (SmallStep-Spawn-Wait) above) and then eliminated. To achieve this "elimination", we may think of adding a new rule, which appears to be so natural that one may even wonder "how did we miss it in our list above?":

$$\langle\, \texttt{spawn}\, s \texttt{ ; skip}\,, \sigma\rangle \to \langle\, \texttt{spawn}\, s, \sigma\rangle$$

This rule turns out to be insufficient and, once we fix the semantics properly, it will actually become unnecessary. Nevertheless, once we add this rule, the resulting small-step SOS can also produce

the behavior in which $x = 1$ at the end of the execution of the second program at the beginning of Section 3.8.4. However, it is still insufficient to produce the behavior in which $x = 100$.

In order to produce a behavior in which $x = 100$ when executing the second program at the beginning of Section 3.8.4, the main thread should first execute its `x + 100` step (which evaluates to 100), then let the two child threads do their writes to x, and then write the 100 to x. We have, unfortunately, no rule that allows computations within s_2 in a sequential composition $s_1 \texttt{;} s_2$ where s_1 is different from `skip` or a `spawn` statement, as it is our case here. What we want is some generalization of the rule (SmallStep-Spawn-Wait) above which allows computations in s_2 whenever it is preceded by a sequence of spawns. On paper definitions, one can do that rather informally by means of some informal side condition saying so. If one needs to be formal, which is a must when one needs to execute the resulting language definitions as we do here, one can define a special sequent saying that a statement only spawns new threads and does nothing else (in the same spirit as the the $C\checkmark$ sequents in Exercise 39), and then use it to generalize the rule (SmallStep-Spawn-Wait) above. However, that would be a rather particular and potentially non-modular (what if later on we add agents or other mechanisms for concurrency?) solution.

Our general solution is to instead enforce the sequential composition of IMP to be structurally associative, using the following structural identity:

$$(s_1 \texttt{;} s_2) \texttt{;} s_3 \equiv s_1 \texttt{;} (s_2 \texttt{;} s_3) \qquad \text{(SmallStep-Seq-Assoc)}$$

That means that the small-step SOS reduction rules now apply *modulo* the associativity of sequential composition, that is, that it suffices to find a structurally equivalent representative of a syntactic term which allows a small-step SOS rule to apply. In our case, the original program is structurally equivalent to one whose first statement is the first `spawn` and whose second statement is the sequential composition of the second `spawn` and the assignment of the main thread, and that structurally equivalent program allows all seven desired behaviors, so the original program also allows them. It is important to understand that we cannot avoid enforcing associativity (or, alternatively, the more expensive solution discussed above) by simply parsing the original program so that we start with a right-associative arrangement of the sequentially composed statements. The problem is that right-associativity may be destroyed as the program executes, for example when applying the true/false rules for the `if` statement, so it needs to be dynamically enforced.

Structural identities are not easy to execute and/or implement, because they can quickly yield an exponential explosion in the number of terms that need to be matched by rules. Since in our particular case we only need the fully right-associative representative of each sequential composition, we can even replace the structural identity above by a small-step SOS rule. The problem with this is that the intended computational granularity of the language is significantly modified; for example, the application of a true/false rule for the conditional statement may trigger as many such artificial rearrangement steps as statements in the chosen branch, to an extent that such rearrangement steps could dominate the total number of steps seen in some computations.

Exercise 114. *Same as Exercise 112, but for small-step SOS instead of big-step SOS. When representing the resulting small-step SOS into rewriting logic, the structural identity can be expressed as a rewriting logic equation, this way capturing faithfully the intended computational granularity of the small-step SOS.* ☆ *When implementing the resulting rewriting logic theory into Maude, this equation can either be added as a normal equation (using* `eq`*) or as an* `assoc` *attribute to the sequential composition construct. The former will only be applied from left-to-right when executed using Maude rewriting and search commands, but that is sufficient in our particular case here.*

MSOS

The small-step SOS above can be straightforwardly turned into an MSOS:

$$\frac{s \to s'}{\texttt{spawn}\, s \to \texttt{spawn}\, s'} \qquad \text{(MSOS-Spawn-Arg)}$$

$$\texttt{spawn skip} \to \texttt{skip} \qquad \text{(MSOS-Spawn-Skip)}$$

$$\frac{s_2 \to s_2'}{\texttt{spawn}\, s_1 \texttt{ ; } s_2 \to \texttt{spawn}\, s_1 \texttt{ ; } s_2'} \qquad \text{(MSOS-Spawn-Wait)}$$

$$(s_1 \texttt{ ; } s_2) \texttt{ ; } s_3 \equiv s_1 \texttt{ ; } (s_2 \texttt{ ; } s_3) \qquad \text{(MSOS-Seq-Assoc)}$$

Even though the MSOS rules above are conceptually identical to the original small-step SOS rules, they are more modular because, unlike the former, they carry over unchanged when the configuration needs to change. Note that, for the same reasons as above, the structural identity is necessary.

Exercise 115. *Same as Exercise 114, but for MSOS instead of small-step SOS.*

Evaluation Contexts

The first and the third rules above are essentially computation propagation rules. In reduction semantics with evaluation contexts the role of such rules is taken over by the splitting/plugging mechanism, which in turn relies on parsing and therefore needs productions for evaluation contexts. We can therefore replace those rules by appropriate productions for evaluation contexts:

$$\textit{Context} \;::=\; \ldots \;\mid\; \texttt{spawn}\, \textit{Context} \;\mid\; \texttt{spawn}\, \textit{Stmt}\, \texttt{;}\, \textit{Context}$$

$$\texttt{spawn skip} \to \texttt{skip} \qquad \text{(RSEC-Spawn)}$$

$$(s_1 \texttt{ ; } s_2) \texttt{ ; } s_3 \equiv s_1 \texttt{ ; } (s_2 \texttt{ ; } s_3) \qquad \text{(RSEC-Seq-Assoc)}$$

The second production for evaluation contexts above involves two language constructs, both the `spawn` and the sequential composition and that the desired non-determinism due to concurrency is captured by deliberate ambiguity in parsing evaluation contexts and, implicitly, in the splitting/plugging mechanism. Note that the structural identity is still necessary.

Exercise 116. *Same as Exercise 114, but for RSEC instead of small-step SOS.*

CHAM

As stated in Section 3.6.1, the CHAM has been specifically proposed as a model of concurrent computation, based on the chemical metaphor that molecules in solutions can get together and react, with possibly many reactions taking place concurrently. Since there was no concurrency so far in our language, the actual strength of the CHAM has not been seen yet. Recall that the configuration of the existing CHAM semantics of IMP consists of one top-level solution, which contains two subsolutions: a syntactic subsolution holding the remainder of the program organized as a molecule sequentializing computation tasks using the special construct $\curvearrowright$; and a state subsolution containing binding molecules, each binding a different program variable to a value. As seen in Figures 3.41 and

3.42, most of the CHAM rules involve only the syntactic molecule. The state subsolution is only mentioned when the language construct involves program variables.

The above suggests that all a `spawn` statement needs to do is to create an additional syntactic subsolution holding the spawned statement, letting the newly created subsolution molecule to float together with the original syntactic molecule in the same top-level solution. Minimalistically, this can be achieved with the following CHAM rule (which does not consider thread termination yet):

$$\{\!|\, \texttt{spawn}\ s \curvearrowright c \,|\!\} \to \{\!|\, \texttt{skip} \curvearrowright c \,|\!\}\ \{\!|s|\!\}$$

Since the order of molecules in a solution is irrelevant, the newly created syntactic molecule "has the same rights" as the original molecule in reactions involving the state molecule. We can rightfully think of each syntactic subsolution as an independently running thread. The same CHAM rules we had before (see Figures 3.41 and 3.42) can now also apply to the newly created threads. Moreover, reactions taking place only in the syntactic molecules, which are a majority by a large number, can apply truly concurrently. For example, a thread may execute a loop unrolling step while another thread may concurrently perform an addition. The only restriction regarding concurrency is that rule instances must involve disjoint molecules in order to proceed concurrently. That means that it is also possible for a thread to read or write the state while another thread, truly concurrently, performs a local computation. This degree of concurrency was not possible within the other semantic approaches discussed so far in this chapter.

The rule above only creates threads. It does not collect threads when they complete their computation. One could do that with the simple solution-dissolving rule

$$\{\!|\, \texttt{skip} \,|\!\} \to \varnothing$$

but the problem is that such a rule cannot distinguish between the original thread and the others, so it would also dissolve the original thread when it completes. However, recall that our design decision for each IMP language extension was to always terminate the program normally, no matter whether it uses the new features or not. The IMP normal result configurations contain a syntactic solution and a state solution, the former holding only `skip`. To achieve that, we can flag the newly created threads for collection as below. Here is our complete CHAM semantics of `spawn`:

Molecule ::= ... | `die`

$$\{\!|\, \texttt{spawn}\ s \curvearrowright c \,|\!\} \to \{\!|\, \texttt{skip} \curvearrowright c \,|\!\}\ \{\!|s \curvearrowright \texttt{die} \,|\!\} \qquad \text{(CHAM-Spawn)}$$

$$\{\!|\, \texttt{skip} \curvearrowright \texttt{die} \,|\!\} \to \varnothing \qquad \text{(CHAM-Die)}$$

Exercise 117. *Same as Exercise 114, but for the CHAM instead of small-step SOS.*

We conclude this section with a discussion on the concurrency of the CHAM above. As already argued, it allows for truly concurrent computations to take place, provided that their corresponding CHAM rule instances do not overlap. While this already goes far beyond the other semantical approaches in terms of concurrency, it still enforces interleaving where it should not. Consider, for example, a global configuration in which two threads are about to lookup two different variables in the state cell. Even though there are good reasons to allow the two threads to proceed concurrently, the CHAM above will not, because the two rule instances (of the same CHAM lookup rule) overlap on the state molecule. This problem can be ameliorated, to some extent, by changing the structure of the top-level configuration to allow all the variable binding molecules currently in the state

subsolution to instead float in the top-level solution at the same level with the threads: this way, each thread can independently grab the binding it is interested in without blocking the state anymore. Unfortunately, this still does not completely solve the true concurrency problem, because one could argue that different threads should also be allowed to concurrently read the same variable. Thus, no matter where the binding of that variable is located, the two rule instances cannot proceed concurrently. Moreover, flattening all the syntactic and the semantic ingredients in a top level solution, as the above "fix" suggests, does not scale. Real-life languages can have many configuration items of various kinds, such as, environments, heaps, function/exception/loop stacks, locks held, and so on. Collapsing the contents of all these items in one flat solution would not only go against the CHAM philosophy, but it would also make it hard to understand and control. The K framework (see Section 5) solves this problem by allowing its rules to state which parts of the matched subterm are shared with and which can be concurrently modified by other rules.